11 | Shuffle的工作原理：为什么说Shuffle是一时无两的性能杀手?

2021-04-07 吴磊

Spark性能调优实战

进入课程 >

讲述：吴磊
时长 21:01 大小 19.26M

你好，我是吴磊。

一提到 Shuffle，你能想到什么？我想很多人的第一反应都是应用中最顽固、最难解决的性能瓶颈。

在之前的课程中，我们也不止一次地提到 Shuffle，尤其是在开发原则那一讲，我还建议你遵循“能省则省、能拖则拖”的原则，在应用中尽量去避免 Shuffle，如果受业务逻辑所限确实不能避免，就尽可能地把 Shuffle 往后拖。

那么，为什么我们一谈 Shuffle 就色变，提到它就避之唯恐不及呢？今天这一讲，我就通过实现一个仙女散花游戏的过程，来和你深入探讨 Shuffle 是如何工作的，说说它为什么

是分布式应用一时无两的性能杀手。毕竟，只有了解 Shuffle 的工作原理，我们才能更好地避免它。

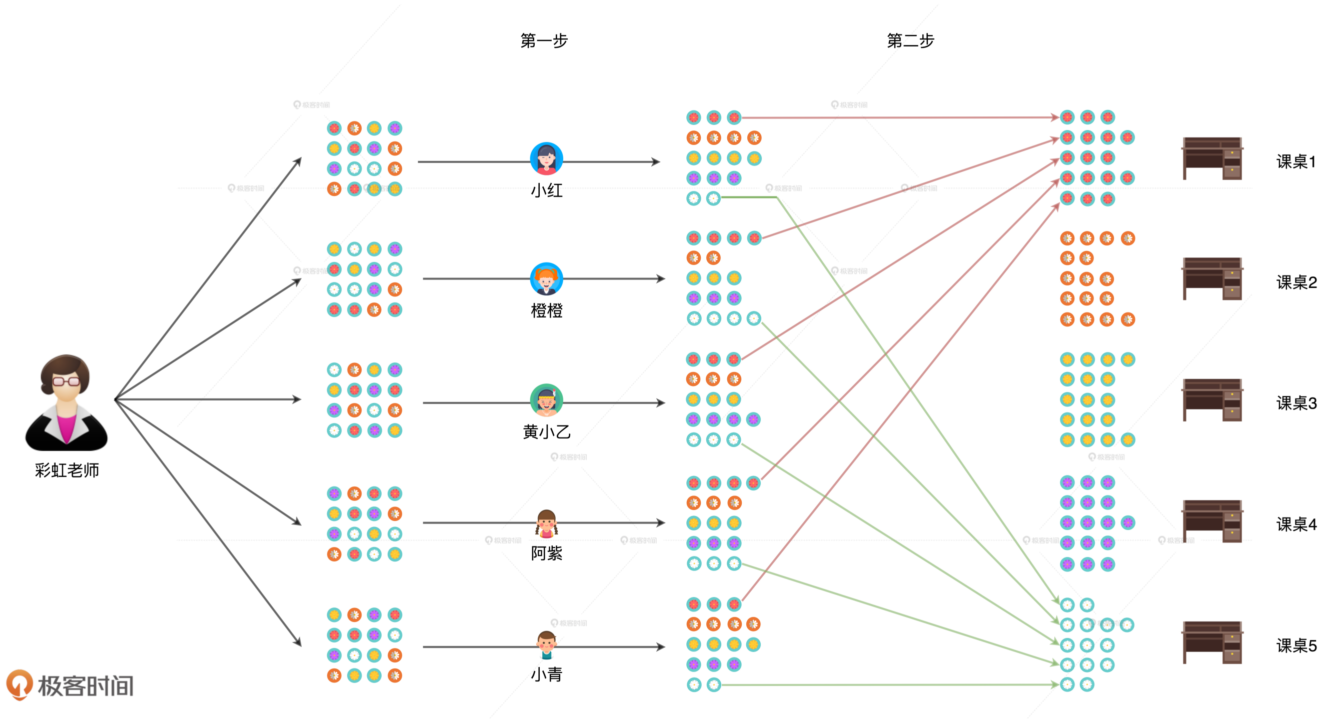
如何理解 Shuffle?

假设，你的老板今天给你安排一个开发任务，让你用 Spark 去实现一个游戏需求。这个实现需求来自一个小故事：仙女散花。

很久以前，燕山脚下有个小村庄，村子里有所“七色光”小学，方圆百里的孩子都来这里上学。这一天，一年级 2 班的黄老师和班里的五个孩子正在做一个游戏，叫做“仙女散花”。

黄老师的背包里装满了五种不同颜色的花朵，五种颜色分别是红、橙、黄、紫、青。她把背包里的花朵随机地分发给五个小同学：小红、橙橙、黄小乙、阿紫和小青。花朵发完之后，每个同学分到的花朵数量差不多，颜色各有不同。

接着，黄老师开始宣布游戏规则：“你们要一起协作，在最短的时间内，把花朵按照颜色收集在一起。游戏完成的标志是，课桌上有 5 堆花朵，每堆花朵的颜色都是一样的。”



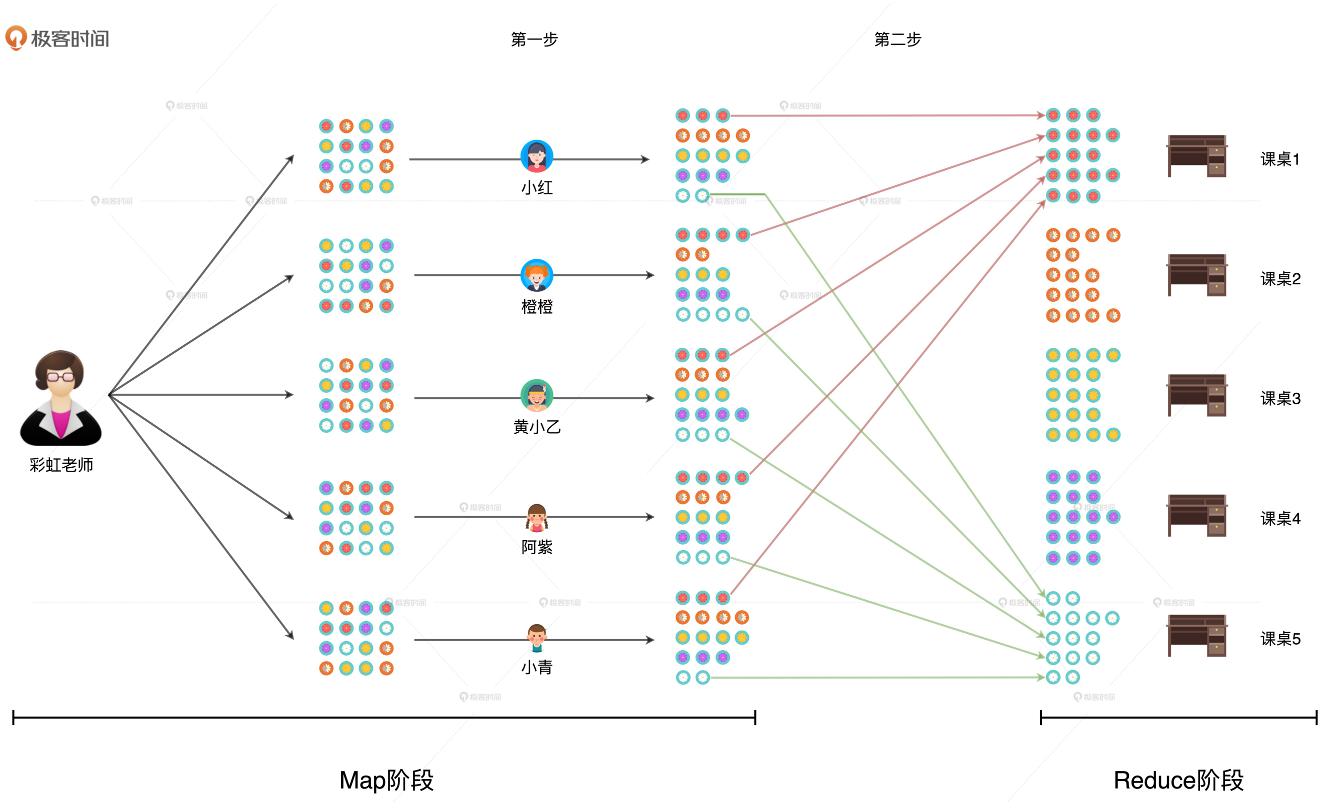
仙女散花的游戏

大家跃跃欲试，黄小乙说：“先别急，我们来制定个策略。先在前面摆上 5 张课桌，然后每个人需要做两件事情，先把自己手里的花朵按照颜色归类分为 5 堆，再把分好颜色的花

朵，分别放到相应的课桌上！”于是，几个小同学按照黄小乙说的策略，不一会儿就完成了游戏。

事实上，仙女散花的游戏过程和 Shuffle 的工作流程大同小异。当然，Shuffle 过程中涉及的步骤和环节，要比“仙女散花”复杂一些。

Shuffle 的本意是“洗牌”，在分布式计算环境中，它有两个阶段。一般来说，前一个阶段叫做“Map 阶段”，后一个阶段叫做“Reduce 阶段”。当然，也有人把它们叫做 Shuffle Write 阶段和 Shuffle Read 阶段。



仙女散花中的Map和Reduce阶段

在仙女散花的游戏中，从老师分发花朵，到 5 个小同学把花朵按照颜色归类，对应的是 Shuffle 的 Map 阶段，而大家把归类的花朵分发到相应的课桌，这个过程类似于 Shuffle 的 Reduce 阶段。

接下来，我们就借助这个故事来深入探讨 Shuffle 的两个阶段：Map 阶段和 Reduce 阶段。

自 2.0 版本之后，Spark 将 Shuffle 操作统一交由 Sort shuffle manager 来管理。因此，今天这一讲，我们专注于 Sort shuffle manager 实现的 Shuffle 分发过程。

Map 阶段是如何输出中间文件的?

以终为始、以结果为导向的学习往往效率更高，在深入探讨 Map 阶段如何生产数据之前，我们不妨先来明确：**Map 阶段的输出到底是什么?**

之前我们也说过，Map 阶段最终生产的数据会以中间文件的形式物化到磁盘中，这些中间文件就存储在 spark.local.dir 设置的文件目录里。中间文件包含两种类型：一类是后缀为 data 的数据文件，存储的内容是 Map 阶段生产的待分发数据；另一类是后缀为 index 的索引文件，它记录的是数据文件中不同分区的偏移地址。这里的分区是指 Reduce 阶段的分区，**因此，分区数量与 Reduce 阶段的并行度保持一致。**

Map阶段输出的数据文件和索引文件

这样一来，我们就可以把问题进一步聚焦在，Spark 在 Map 阶段是如何生产这些中间文件的。不过，我们首先需要明确的是，Map 阶段每一个 Task 的执行流程都是一样的，每个 Task 最终都会生成一个数据文件和一个索引文件。**因此，中间文件的数量与 Map 阶段的并行度保持一致。**换句话说，有多少个 Task，Map 阶段就会生产相应数量的数据文件和索引文件。

接下来，我带你用 Spark 来实现“仙女散花”的游戏，咱们一边做游戏，一边来分析 Map 阶段的中间文件是如何产生的。

用 groupByKey 实现“仙女散花”

在“仙女散花”的游戏中，黄老师要求大家把同一种花色的花朵**收集**到一起。那么，在 Spark 的分布式开发框架内，与这个游戏最相仿的计算过程非 groupByKey 莫属，所以我们不妨用 groupByKey 来实现游戏。

首先是 flowers.txt 文件：

复制代码

```
黄色花朵
紫色花朵
红色花朵
橙色花朵
青色花朵
黄色花朵
紫色花朵
橙色花朵
青色花朵
......

```

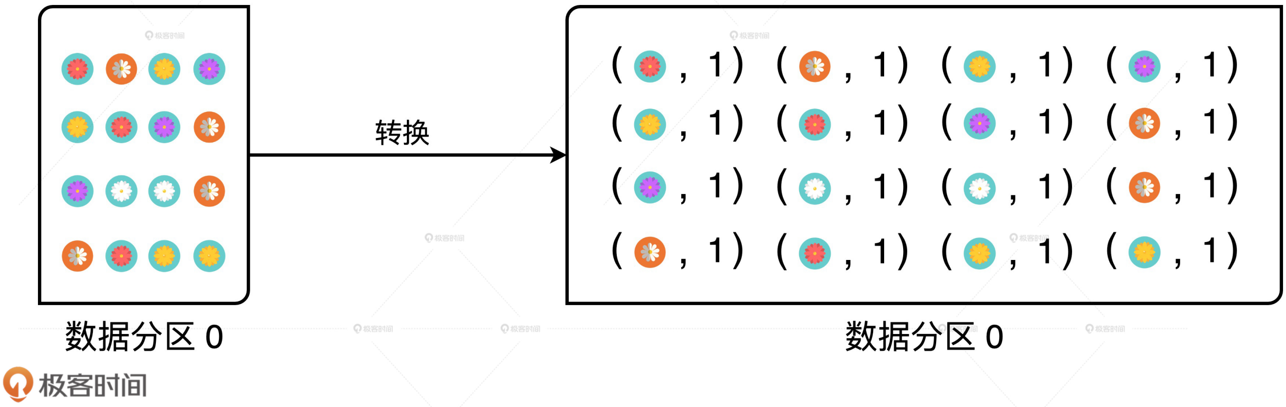
其次是同学小 A 接到需求后，用 groupByKey 实现“仙女散花”游戏的代码：

复制代码

```
val flowers = spark.sparkContext.textFile("flowers.txt")
//黄老师给5个小同学分发花朵
val flowersForKids = flowers.coalesce(5)
val flowersKV = flowersForKids.map((_, 1))
//黄小乙的两个步骤：大家先各自按颜色归类，然后再把归类后的花朵放到相应的课桌上
flowersKV.groupByKey.collect

```

我们可以看到，代码步骤与游戏过程基本上一一对应。但是，读取完花朵文件之后，由于 groupByKey 是 pairRDD 算子，需要消费（Key，Value）形式的数据，因此我们需要对原始花朵数据做一次转换。以数据分区 0 为例，数据的转换过程如下图所示，你不妨把数据分区 0 理解为是黄老师分发给小红的花朵。

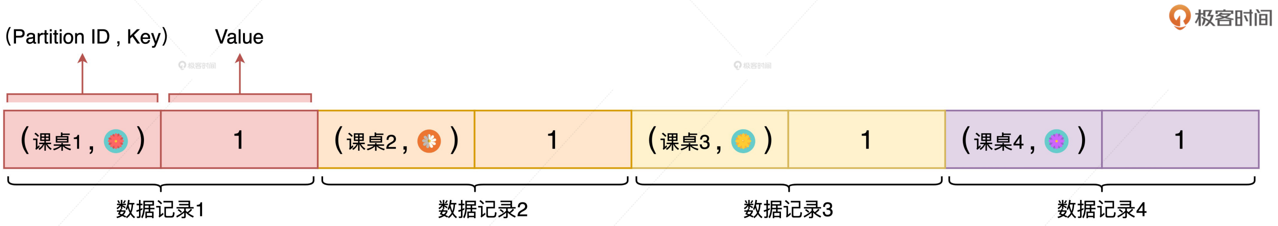


将原始数据转换为pairRDD

基于 pairRDD 的 Key，也就是花朵的颜色，Map Task 就可以计算每条数据记录在 Reduce 阶段的目标分区，目标分区也就是游戏中的课桌。在黄小乙制定的策略中，哪种花放到哪张桌子是大家事先商定好的，但在 Spark 中，每条数据记录应该分发到哪个目标分区，是由 Key 的哈希值决定的。

目标分区计算好之后，Map Task 会把每条数据记录和它的目标分区，放到一个特殊的数据结构里，这个数据结构叫做“PartitionedPairBuffer”，它本质上就是一种数组形式的缓存结构。它是怎么存储数据记录的呢？

每条数据记录都会占用数组中相邻的两个元素空间，第一个元素是（目标分区，Key），第二个元素是 Value。假设 PartitionedPairBuffer 的大小是 4，也就是最多只能存储 4 条数据记录。那么，如果我们还以数据分区 0 为例，小红的前 4 枚花朵在 PartitionedPairBuffer 中的存储状态就会如下所示。



PartitionedPairBuffer存储小红的前4枚花朵

对我们来说，最理想的情况当然是 PartitionedPairBuffer 足够大，大到足以容纳 Map Task 所需处理的所有数据。不过，现实总是很骨感，每个 Task 分到的内存空间是有限的，PartitionedPairBuffer 自然也不能保证能容纳分区中的所有数据。因此，**Spark 需要**

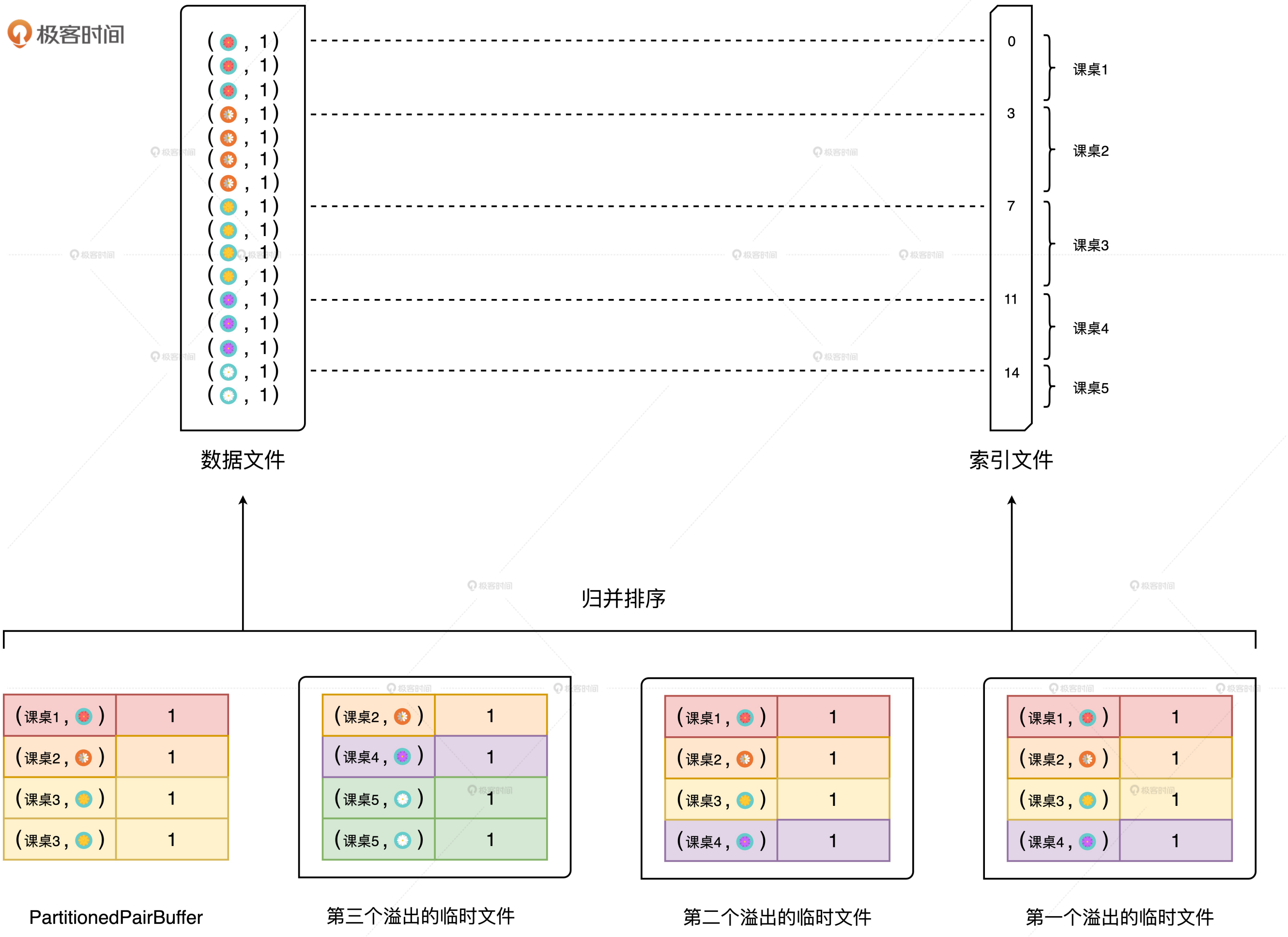
一种计算机制，来保障在数据总量超出可用内存的情况下，依然能够完成计算。这种机制就是：排序、溢出、归并。

就拿大小为 4 的 PartitionedPairBuffer 来说，数据分区 0 里面有 16 朵花，对应着 16 条数据记录，它们至少要分 4 批才能依次完成处理。在处理下一批数据之前，Map Task 得先把 PartitionedPairBuffer 中已有的数据腾挪出去，腾挪的方式简单粗暴，Map Task 直接把数据溢出到磁盘中的临时文件。

不过，在溢出之前，对于 PartitionedPairBuffer 中已有的数据，Map Task 会先按照数据记录的第一个元素，也就是目标分区 + Key 进行排序。也就是说，尽管数据暂时溢出到了磁盘，但是临时文件中的数据也是有序的。

就这样，PartitionedPairBuffer 腾挪了一次又一次，数据分区 0 里面的花朵处理了一批又一批，直到所有的花朵都被处理完。分区 0 有 16 朵花，PartitionedPairBuffer 的大小是 4，因此，PartitionedPairBuffer 总共被腾挪了 3 次，生成了 3 个临时文件，每个临时文件中包含 4 条数据记录。16 条数据，有 12 条分散在 3 个文件中，还有 4 条缓存在 PartitionedPairBuffer 里。

到此为止，我们离 Map 阶段生产的、用于在网络中分发数据的中间文件仅有一步之遥了。还记得吗？Map 阶段生产的中间文件有两类，一类是数据文件，另一类是索引文件。分散在 3 个临时文件和 PartitionedPairBuffer 里的数据记录，就是生成这两类文件的输入源。最终，Map Task 用归并排序的算法，将 4 个输入源中的数据写入到数据文件和索引文件中去，如下图所示。



归并临时文件，生成最终的中间文件

好了，到目前为止，我们用 groupByKey 实现了“仙女散花”的游戏，详细讲解了 Map 阶段生产中间文件的过程。虽然 Map 阶段的计算步骤很多，但其中最主要的环节可以归结为 4 步：

1. 对于分片中的数据记录，逐一计算其目标分区，并将其填充到 PartitionedPairBuffer；

2. PartitionedPairBuffer 填满后，如果分片中还有未处理的数据记录，就对 Buffer 中的数据记录按（目标分区 ID，Key）进行排序，将所有数据溢出到临时文件，同时清空缓存；

3. 重复步骤 1、2，直到分片中所有的数据记录都被处理；

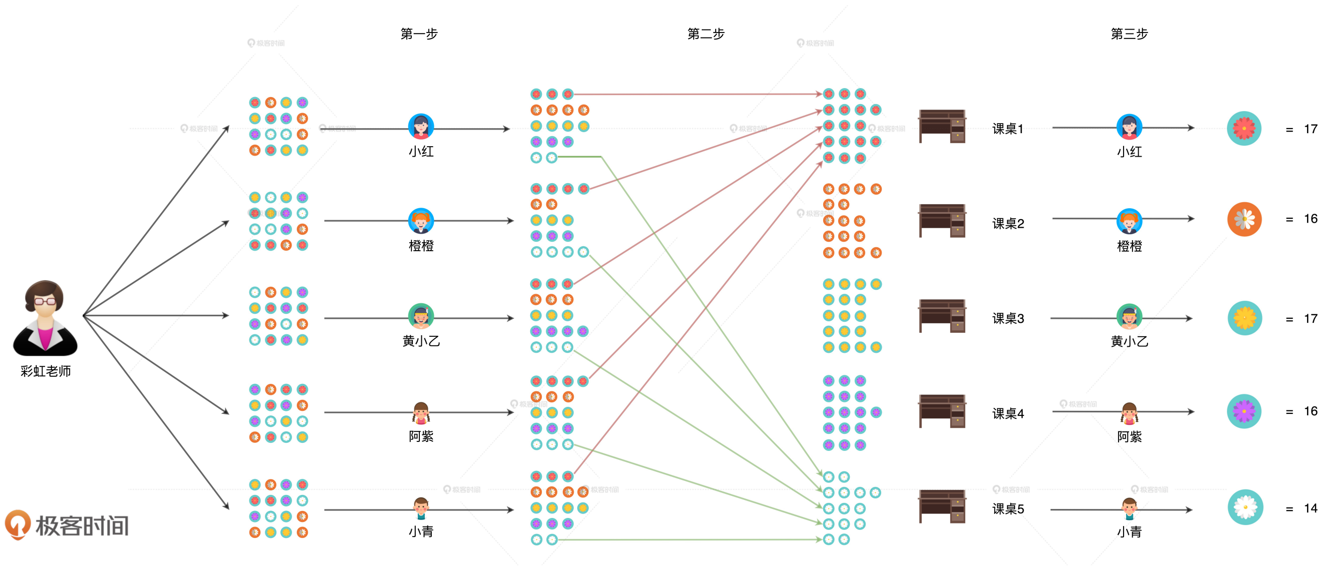
4. 对所有临时文件和 PartitionedPairBuffer 归并排序，最终生成数据文件和索引文件。

不难发现，仙女散花其实就是个分组、收集的游戏。应该说，用 Spark 来实现分组、收集类的游戏还是比较简单的，那么，如果把仙女散花变成是“分组、统计”的游戏，我们该怎么做呢？

“仙女散花”游戏升级

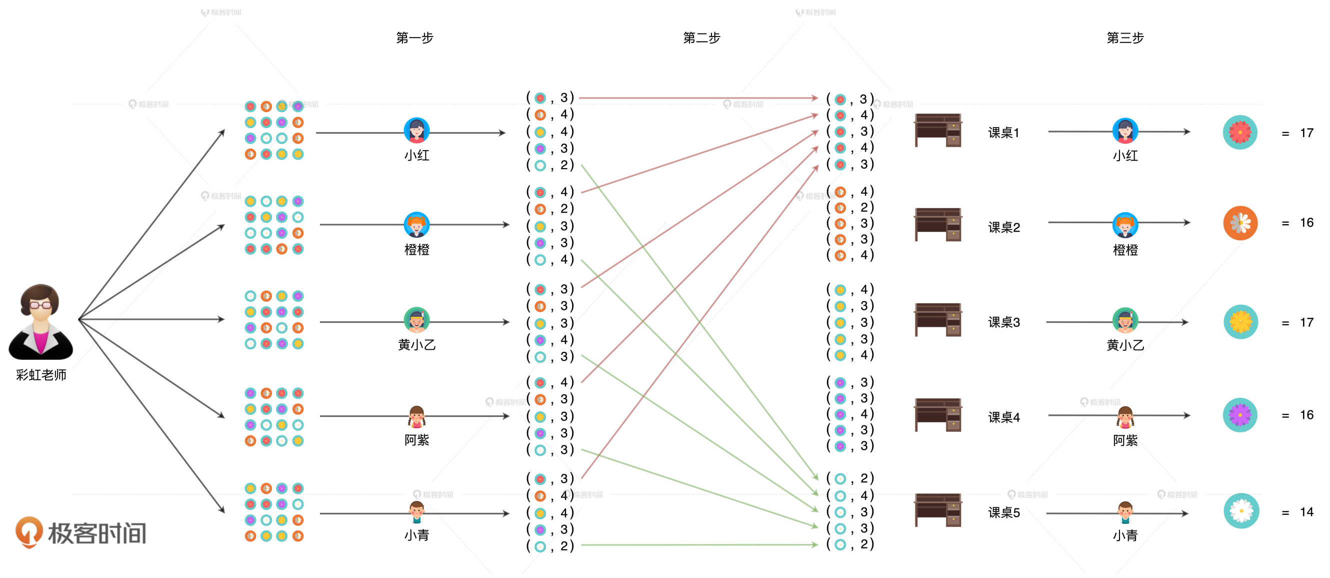
5 个小同学完成游戏之后，离下课时间还早。因此，黄老师调整了游戏规则：“你们五个人还是一起协作，这次要在最短的时间内，统计不同花色花朵的数量。”

小红迫不及待地说：“很简单！还是按照刚才的策略，先把花朵分好堆，然后我们五个人分别去课桌上数数就好啦！”



小红的主意

黄小乙皱了皱眉，说道：“别急，新的游戏规则也是有时间限制的，我想了一个和你差不多的办法，一共分三步：第一步，每个人把手里不同花色花朵的数量先算出来；第二步，我们只需要把花朵的数量写到相应的桌子上；第三步，我们分别对五张课桌上的数字求和。这样就能完成得更快了”



黄小乙的主意

用 reduceByKey 实现升级后的仙女散花

如果我们想用 Spark 来实现升级后的游戏，该怎么办呢？其实很简单，只要把 groupByKey 换成 reduceByKey 就好了。

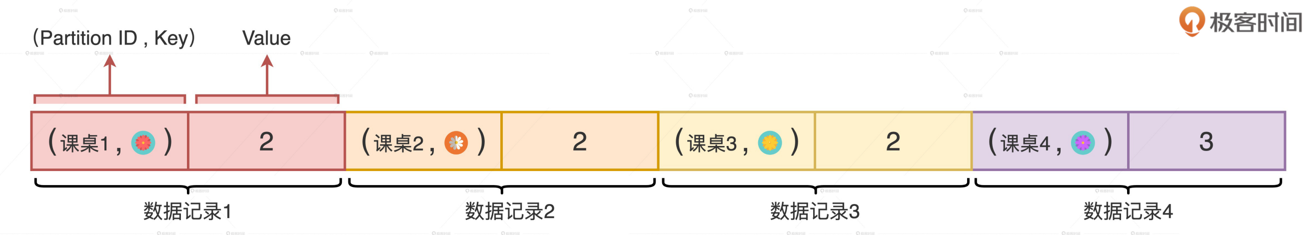
复制代码

```
val flowers = spark.sparkContext.textFile("flowers.txt")
//黄老师给5个小同学分发花朵
val flowersForKids = flowers.coalesce(5)
val flowersKV = flowersForKids.map((_, 1))
//黄小乙的两个步骤：大家先各自按颜色计数，然后再按照课桌统一计数
flowersKV.reduceByKey(_ + _).collect

```

接下来，我们来分析一下 reduceByKey 的 Map 阶段计算，相比 groupByKey 有何不同。就 Map 端的计算步骤来说，reduceByKey 与刚刚讲的 groupByKey 一样，都是先填充内存数据结构，然后排序溢出，最后归并排序。

区别在于，在计算的过程中，**reduceByKey 采用一种叫做 PartitionedAppendOnlyMap 的数据结构来填充数据记录**。这个数据结构是一种 Map，而 **Map 的 Value 值是可累加、可更新的**。因此，PartitionedAppendOnlyMap 非常适合聚合类的计算场景，如计数、求和、均值计算、极值计算等等。



大小为4的PartitionedAppendOnlyMap

在上图中，4 个 KV 对的 Value 值，是扫描到数据分区 0 当中青色花朵之前的状态。在 PartitionedAppendOnlyMap 中，由于 Value 是可累加、可更新的，因此这种数据结构可以容纳的花朵数量一定比 4 大。因此，相比 PartitionedPairBuffer，PartitionedAppendOnlyMap 的存储效率要高得多，溢出数据到磁盘文件的频率也要低得多。

以此类推，最终合并的数据文件也会小很多。**依靠高效的内存数据结构、更少的磁盘文件、更小的文件尺寸，我们就能大幅降低了 Shuffle 过程中的磁盘和网络开销。**

事实上，相比 groupByKey、collect_list 这些收集类算子，聚合类算子（reduceByKey、aggregateByKey 等）在执行性能上更占优势。**因此，我们要避免在聚合类的计算需求中，引入收集类的算子。**虽然这种做法不妨碍业务逻辑实现，但在性能调优上可以说是大忌。

Reduce 阶段是如何进行数据分发的?

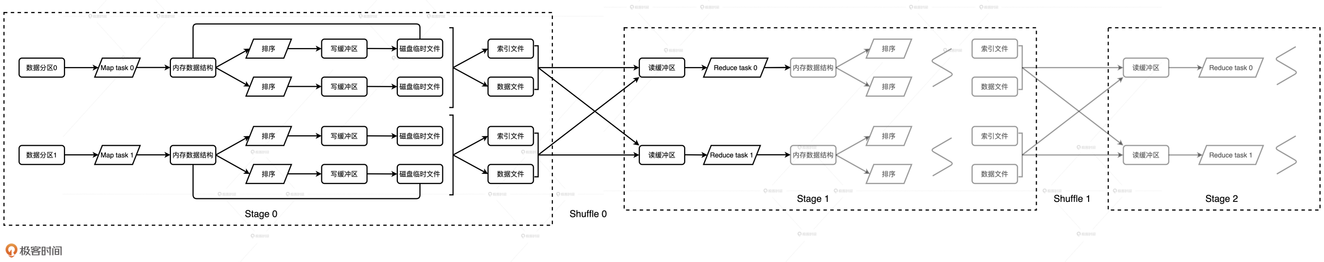
最后，我们再来说说 Reduce 阶段，在“仙女散花”的游戏里，每个人把自己的花朵归好类之后，主动地把不同颜色的花朵放到相应的课桌上，这个过程实际上就是 Shuffle 过程中的数据分发。不过，与课桌被动地接收花朵不同的是，Shuffle 在 Reduce 阶段是主动地从 Map 端的中间文件中拉取数据。

Map阶段输出的数据文件和索引文件

刚刚讲过，每个 Map Task 都会生成如上图所示的中间文件，文件中的分区数与 Reduce 阶段的并行度一致。换句话说，每个 Map Task 生成的数据文件，都包含所有 Reduce Task 所需的部分数据。因此，任何一个 Reduce Task 要想完成计算，必须先从所有 Map Task 的中间文件里去拉取属于自己的那部分数据。索引文件正是用于帮助判定哪部分数据属于哪个 Reduce Task。

Reduce Task 通过网络拉取中间文件的过程，实际上就是不同 Stages 之间数据分发的过程。在“仙女散花”的游戏中，5 个孩子与 5 张课桌之间，需要往返 25 人次。如果让 100 个孩子把 100 种颜色的花朵，分别收集到 100 张课桌上，那么这 100 个孩子与 100 张课桌之间，就需要 10000 人次的往返！显然，Shuffle 中数据分发的网络开销，会随着 Map Task 与 Reduce Task 的线性增长，呈指数级爆炸。

Reduce Task 将拉取到的数据块填充到读缓冲区，然后按照任务的计算逻辑不停地消费、处理缓冲区中的数据记录，如下图所示。



Reduce阶段的计算过程

我们可以看到，Reduce 阶段用圆圈标记的 1、2、3、4 与 Map 阶段的四个步骤一模一样。没错，因即是果、果即是因，当我们说某个 Stage 是 Map 阶段或是 Reduce 阶段的时候，我们的出发点或者说锚点就是 Shuffle。对于上图的 Shuffle 0 来说，Stage 0 是 Map 阶段，Stage 1 是 Reduce 阶段。但是，对于后面的 Shuffle 1 来说，Stage 1 就变成了 Map 阶段。因此你看，当我们把视角拉宽，Map 和 Reduce 这两个看似对立的东西，其实有很多地方都是相通的。

性能杀手

想必经过上面两个阶段的分析，你已经对 Shuffle 为何会成为性能瓶颈，有了比较直观的感受。这里，我再带你总结一下。

首先，对于 Shuffle 来说，它需要消耗所有的硬件资源：

- 无论是 PartitionedPairBuffer、PartitionedAppendOnlyMap 这些内存数据结构，还是读写缓冲区，都会消耗宝贵的内存资源；
- 由于内存空间有限，因此溢出的临时文件会引入大量磁盘 I/O，而且，Map 阶段输出的中间文件也会消耗磁盘；
- 呈指数级增长的跨节点数据分发，带来的网络开销更是不容小觑。

其次，Shuffle 消耗的不同硬件资源之间很难达到平衡。磁盘和网络的消耗是 Shuffle 中必需的环节。但是，磁盘与网络的处理延迟相比 CPU 和内存要相差好几个数量级。以下表为例，如果以 CPU L1 缓存的处理延迟为基准，把单位从纳秒校准到秒，我们就会惊讶地发现，当 CPU、内存以秒为单位处理数据时，磁盘和网络的处理延迟是以天、月为单位的！

硬件资源	处理延迟	时间单位换算
CPU L1 缓存	0.5纳秒	0.5秒
CPU L2 缓存	7纳秒	7秒
内存	100纳秒	100秒
内存顺序读(1MB)	250微秒	2.9天
同DC网络传输	0.5毫秒	5.8天
HDD顺序读(1MB)	20毫秒	7.8月

极客时间

正是基于 Shuffle 的这些特点，我们才会“谈虎色变”，一提到 Shuffle 就避之唯恐不及，强调能省则省、能拖则拖。

小结

这一讲，我借助实现仙女散花这个游戏的需求，带你直观地认识 Shuffle 的计算过程。Shuffle 有两个计算阶段，Map 阶段和 Reduce 阶段。我们要重点掌握 Map 阶段的计算流程，我把它总结为 4 步：

1. 对于分片中的数据记录，逐一计算其目标分区，然后填充内存数据结构（PartitionedPairBuffer 或 PartitionedAppendOnlyMap）；
2. 当数据结构填满后，如果分片中还有未处理的数据记录，就对结构中的数据记录按（目标分区 ID，Key）排序，将所有数据溢出到临时文件，同时清空数据结构；
3. 重复前 2 个步骤，直到分片中所有的数据记录都被处理；
4. 对所有临时文件和内存数据结构中剩余的数据记录做归并排序，最终生成数据文件和索引文件。

在 Reduce 阶段我们要注意，Reduce Task 通过网络拉取中间文件的过程，实际上就是不同 Stages 之间数据分发的过程。并且，Shuffle 中数据分发的网络开销，会随着 Map Task 与 Reduce Task 的线性增长，呈指数级爆炸。

最后，从硬件资源的角度来看，Shuffle 对每一种硬件资源都非常地渴求，尤其是内存、磁盘和网络。由于不同硬件资源之间的处理延迟差异巨大，我们很难在 Shuffle 过程中平衡

CPU、内存、磁盘和网络之间的计算开销。因此，对于 Shuffle 我们避之唯恐不及，要能省则省、能拖则拖。

每日一练

1. 以小红分到的花朵（数据分区 0）为例，你能推导出 reduceByKey 中 Map 阶段的每个环节吗？（提示：PartitionedAppendOnlyMap 需要多少次溢出到磁盘临时文件？每一个临时文件中的内容是什么？最终生成的中间文件，内容分别是什么？和 groupByKey 生成的中间文件一样吗？）
2. Map 阶段和 Reduce 阶段有不少环节都涉及数据缓存、数据存储，结合上一讲介绍的 Spark 配置项，你能把相关的配置项对号入座吗？

期待在留言区看到你的思考和讨论，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 |调优一筹莫展，配置项速查手册让你事半功倍！（下）

下一篇 12 | 广播变量（一）：克制Shuffle，如何一招制胜！

精选留言 (6)

写留言

对方正在输入。。。
2021-04-13

老师我补充一下，采用sortShuffle的方式时，只有满足在shuffleDependency里面aggeragator或者sort这两个字段有效时，才会根据partitionid和key排序，否则只根据partitionid排序。如不会按照key排序的算子有repartition

展开

作者回复: 是的，没错！补充的很到位，谢谢哈~

5

苏子浩

2021-04-19

老师我想确认一下，问题一所提到的reduceByKey 中 Map 阶段每一个临时文件中的内容是否和该对应数据分区中的数据记录的顺序有关呢？因为分区中数据记录的不同而导致Map 阶段每一个临时文件中的内容不同。根据在文中提到的GroupBuKey算子的计算步骤二：“PartitionedPairBuffer 填满后，如果分片中还有未处理的数据记录，就对 Buffer 中的数据记录按（目标分区 ID，Key）进行排序，将所有数据溢出到临时文件，同时清...

展开

作者回复: 非常赞~ 👍 思考的很到位，没错，就像你说的那样~

首先，溢出的临时文件的内容，就是和分区数据的顺序有关，你说的没错，同样的分片，换个顺序，溢出的文件内存（对reduceByKey来说）大概率是不同的；

再者，不论溢出文件的内容相同与否，同样的分片数据，最终的中间文件是一致的，这个是由最终的Merge Sort决定的。

最后，就像你分析的，reduceByKey溢出的临时文件，一定会比groupByKey更少，原因很简单，就像你说的，因为有Map端聚合，所以自然节省空间，需要溢出的频率就降低了，溢出文件自然更少。

很棒~ 耐心地推导了reduceByKey的全部过程，赞赞赞~

2 2

井先生

2021-04-08

答题1：

溢写出一个文件

[（分区id，红），3] ,[（分区id，橙），4] ,[（分区id，黄），4] ,[（分区id，紫），3]

partitionedAppendOnlyMap里面的数据

[（分区id，蓝），2] ...

展开

作者回复: 第一题满分💯 ~

map端聚合，主要是降低数据量，减少磁盘和网络的开销。归并排序主要是保证运行的稳定性，因为归并排序可以通过磁盘来做，也就是用External Sorter来做这事，这样不会因为内存受限而频繁地oom。

1 2

巧克力黑
2021-04-15

老师好，这节学习过后生成许多疑问，请老师帮忙解惑
想就“千女散花”的过程详细了解下spark.shuffle.sort.bypassMergeThreshold
1、“千女散花”过程中的groupByKey和reduceByKey是否都不需要引入排序操作？ 目前我觉得是不需要
...
展开

作者回复: 好问题，我们一个个来看。

1. 确实都不需要排序，但是，在Spark的byPass机制下，groupByKey可以通过设置spark.shuffle.sort.bypassMergeThreshold从而跳过排序操作，但是，reduceByKey不行，因为它有Aggregate聚合操作，所以，坦白说，Sort shuffle manager对于reduceByKey这种操作确实不太公平，但是没办法，它的实现机制就是这样。

2. 没错，在默认排序的情况下，你说的这几个步骤，都会有排序，都会受排序影响。

3. spark.shuffle.sort.bypassMergeThreshold这个参数是控制shuffle的，所以不区分是rdd还是DataFrame、Dataset，如果满足byPass条件，都会生效的~

2 1

October
2021-04-10

Reduce极端的读缓存指的是什么？
展开

作者回复: 老弟，能说的更具体点吗？我都不记得我这么说过[Facepalm]

Fendora范东_
2021-04-07

Map端配置
spark.shuffle.file.buffer 适当调大可以降低落盘次数，以及降低归并排序路数
reduce端配置
spark.reducer.maxSizeInFlight 可以适当调大减少网络io次数
...
展开

作者回复: 对，不是越大越好，你前面分析的很到位～ 如果是针对这两个参数调整，不妨以50%～100%步进式地调高缓存，对比执行性能变化。当然，这里技巧就比较灵活了，如果这块确实是瓶颈，还可以用二分查找。

1